Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 161 (2006) 91–107

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

On Formal Specification of Software Components and Systems

Sharon Flynn

National University of Ireland, Galway

Dick Hamlet[1]

Portland State University, USA

Abstract

Reasoning about software systems developed using components begins with component-level specifications, from which system-level specifications are derived. While sound compositional reasoning is a strength of formal specification methods, practical experience with systems construction leads us to expect surprises when two components that were never intended to be combined are composed. Component specifications, like any other human artifact, are likely to be in error. Composition throws the mistakes in one component against those in another, leading to unexpected and often bizarre behavior.
We review the theory of formal software specification and apply it to the combination of component specifications into system-level properties, where deficiencies in component specifications can be strangely reflected. We conclude that desirable properties of system specifications do not always arise from those same properties at the component level.

Keywords: Formal specification, software component, composition

1 Introduction

Although elementary formal descriptions of program semantics, specifications, and the correctness relationship between them have long been used, applying these descriptions to software components and their combination into systems has been little explored. Furthermore, the role played by persistent state in software has not been singled out for formal analysis. State plays a central role in component-based system development (CBSD), so CBSD is a good setting for a theoretical treatment.

[1] Supported by NSF grant CCR-0112654 and by an E.T.S. Walton fellowship from Science Foundation Ireland. Neither institution is responsible for statements made in this paper.

doi:10.1016/j.entcs.2006.04.027

1.1 Basic Definitions

A *software component* is loosely defined as any executable unit described only by its interface (syntax) and black-box behavior (semantics). This definition (Szyperski [12]) cuts through a great deal of unprofitable controversy about the role of particular programming languages and design methods in CBSD.

Real components are often combined by concurrent execution in which they communicate asynchronously. Temporal logics best capture parallel execution [2] and model checking can be used to reason about component- and system behavors [6]. In contrast we have chosen to address only the sequential, functional aspect of behavior. The rationale for our choice is its mathematical simplicity and the existence of a body of program-analysis theory based on functional abstraction. We hope that the older, non-temporal formalism will illuminate basic issues.

Definition 1.1 Component code is identified with two mappings it computes:

$$f : D \times H \to R,$$

the *functional mapping* from its *input* domain D and *state* domain H to its *output* range R; and

$$g : D \times H \to H,$$

the *state mapping*.

Throughout this paper the symbols D, H, R, f, g will be reserved for their meanings in Definition 1.1.

The intuitive difference between D and H in Definition 1.1 is that the input variable is 'independent' while the state variable is 'dependent.' A program is given an arbitrary input; on the contrary, it itself sets the state that later influences it.

The program semantic mappings are in general partial functions, because it may happen that code does not terminate for some particular input x and state h, so that f and g are not defined at (x, h). We are only occasionally concerned with termination, so unless otherwise explicitly stated, we assume that f and g are total.

In Definition 1.1, a persistent state set H and program behavior g mapping onto it are singled out notationally, which is unusual. The usual view is to treat the program mapping as a single function applying to a wider set which includes input and state values, and also values of internal variables used by the program. It is our contention that state is the source of many difficulties in formal software description, and so should be highlighted. We could do this by defining a single semantic function for a program, then projecting onto a special set for the state mapping. However, such a formalism makes it easier not to notice state as a separate entity, and adds an extra layer of notation (for the projections) whenever state is considered.

The state set is thought of as local to a program and not accessible to any other program. (This becomes significant when we consider combining components in Section 3.) Private state of this kind has two distinct aspects. One, the *concrete*

state, is represented by H itself and is directly manipulated by the code function g. A second set is the *abstract state* J, usually a mathematically defined entity, which enters into specifications. The reason for making a distinction is that J may be a high-level description of the intuitive state, which may not be available as a data type in the programming language. It is then necessary to *represent* values of J by program entities in H. The connection between the two is established by the *abstraction map* $L : H \to J$. This process of representation and abstraction is the basis for information hiding, a practical design technique of the first importance. However, distinguishing abstract state adds little to the situations we wish to discuss, so L will usually be taken to be the identity function.

In principle, specifications need not concern themselves with software state. To describe what is required of a program does not necessarily require a description of persistent storage it will maintain. Indeed, it would be ideal if the decision to use persistent storage at all and its form if used were left to the implementor just as choices about temporary storage (internal variables) are[2]. If the specification does mention state, it could be in the sense of non-binding operational specification: "This is one way to accomplish what is required, but any means that accomplishes the same thing is acceptable." For informal specifications it may be possible to avoid describing state explicitly by using such circumlocutions as, "if this value has been seen N times on previous runs, then print the value of N...". However, it sometimes seems impossible to give a precise, formal description of required actions that depend on previous history without explicitly describing state to capture that history. Hence specification languages and formal methods do incorporate state. One of the issues considered in this paper will be the degree to which state should explicitly enter formal specification.

We begin with a definition that includes state.

Definition 1.2 A *specification* S for code is a relation on $D \times H \times H \times R$.

In Definition 1.2 the first H set in the cross product is intuitively an input state, and the second an output state. Specifications may not notationally distinguish between 'inputs' and 'outputs.' However, the intuition captured in Definition 1.2 is that values in the final two sets in the cross product are prescribed by S, while values in the first two sets are not prescribed. The 'input' state is a particularly touchy quantity, since constraints on the 'output' state also implicitly constrain the 'input.' It is the definition of correctness that captures these input/output distinctions.

The simplest definition is:

Definition 1.3 A program described by functional mapping f and state mapping g is *state-blind correct* wrt S iff $\forall\, x \in D, \forall\, h \in H$, either[3]:

$$(x, h, g(x, h), f(x, h)) \in S, \text{ or}$$

[2] If persistent storage is actually a program output, to be used by other programs, its form does of course need to be prescribed.

[3] If states were abstracted by a mapping $L : H \to J$, the specification would be a relation on $D \times J \times J \times R$ and the definition would read: for every $x \in D$ and every $h \in H$, either $(x, L(h), L(g(x, h), f(x, h)) \in S$, or there is no $r \in R$ such that $(x, L(h), h', r) \in S$ for any h'.

$$\forall\, r \forall\, h', (x, h, h', r) \notin S.$$

In the latter situation, x is called a *don't-care* input in state h.

The intent of a don't-care input x is to use a counter-factual conditional to allow a correct program to take an arbitrary action on x.

Definition 1.3 does not assign state any special role. To do better, we must consider sequences of inputs and the sequences of state values that result. Let $h_0 \in H$ be a special *initial state*, and consider a sequence of inputs $t = (x_0, x_1, ..., x_n)$, $x_i \in D, 0 \leq i \leq n$. Define

$$h_i = g(x_{i-1}, h_{i-1}),\ \ 1 \leq i \leq n.$$

Then successive functional outputs of the program are

$$f(x_0, h_0), f(x_1, h_1), ..., f(x_n, h_n).$$

A definition of correctness for sequence t that captures our intuition about state is:

Definition 1.4 f and g are *correct for sequence* t wrt S iff for all members of t either

$$(x_i, h_i, g(x_i, h_i), f(x_i, h_i)) \in S$$

or x_i is a don't-care input in state h_i. The program is *sequence correct* wrt S iff it is correct for all such sequences.

Theorem 1.5 *State-blind ⇒ sequence correctness, but not the reverse.*

Proof. It is obvious that state-blind correctness implies sequence correctness, since each of the particular relational elements required by the latter are included in the former. To see that the reverse implication does not hold, consider a program P_0 with the description:

$$\begin{aligned} &\forall\, x \forall\, h,\ \ g(x, h) = h_0, \\ &\quad \forall\, x,\ \ f(x, h_0) = 0, \\ &\quad\ \forall\, x,\ \ f(x, h) = 1,\ \ h \neq h_0; \end{aligned}$$

and a specification:

$$\forall\, x, \forall\, h,\ \ (x, h, h, 0) \in S.$$

P_0 is sequence correct wrt S, but not state-blind correct. □

Evidently, state-blind correctness asks more than intuitively necessary; sequence correctness is enough to guarantee intuitively good behavior from a program.

A definition can be framed as a variant of sequence correctness that leaves state entirely to the implementation.

Definition 1.6 Let D^+ be all finite sequences in D^n for any integer $n \geq 1$. Alter Definition 1.2 of a specification S to a relation on $D^+ \times R$, that is, concerned only with output for an input sequence. S must include the definition of a special condition, being *reset* so that one may speak of an input received in this condition, intuitively corresponding to a program in its initial state. The successive functional and state values computed by a program beginning with its initial state on input sequence $t \in D^n$ are as in Definition 1.4, and we could say that the program is correct iff for all t, we have $(t, f(x_n, h_n)) \in S$, where S is reset at the beginning of the sequence.

Call the notions of Definition 1.6 *state-hidden* and in contrast call those in Definitions 1.2 and 1.3 and 1.4 *state-explicit*. State-hidden ideas based on behavior sequences can also be described by temporal logics. This application is distinct from capturing concurrent behavior, but we do not pursue it here. Let S be a state-explicit specification and S' a state-hidden specification. There cannot be a sense in which S and S' are 'equivalent,' since state unlike that prescribed for the former (indeed, perhaps no state at all) may be used to implement the latter. However, we can define the relationship in one direction:

Definition 1.7 S' *covers* S iff all programs that are sequence correct for S are state-hidden correct for S'.

In the sequel, when 'correct' is not qualified, it will mean state-explicit state-blind correct. However, most intuitive interpretations have the same significance for the variant definitions. For example, sequence correctness merely restricts intuitive statements about 'all states' to apply only to states that arise in actual sequences.

1.2 'Wrong' Specifications

A specification S may be in error for some person because:

(i) S does not correspond to the desires of that person, i.e., there is some (x, h) of interest and some E that is unacceptable as an output and/or e unacceptable as a result state on input x in state h, yet $(x, h, e, E) \in S$. We say that S *is incorrect*, or S *is wrong at* (x, h).

(ii) S is *incomplete at* (x, h), meaning that an output on input x in state h is desired by the person, yet x is a don't-care input in state h of S.

(iii) S is *overly prescriptive at* (x, h), meaning that an output r and result state h' on input x in state h is acceptable to the person, yet $(x, h, h', r) \notin S$.

(iv) (Sequence vs. state-blind correctness) S is wrong or incomplete or overly prescriptive at (x, q), but according to S, q should not arise in any input sequence beginning with reset. This situation results from the person being unclear about which kind of correctness is appropriate.

(v) (State-hidden vs. state-explicit correctness) state-explicit S is wrong or incomplete or overly prescriptive at (x, q), yet there exists a state-hidden specification S' covering S for which these errors make no sense. Again, the person is

confusing definitions.

It is conventional (though perhaps not wise) to omit mention of the crucial person when using these terms.

The relationship between over-prescription and incompleteness is somewhat peculiar. On the one hand, it is desirable not to over-specify, that is, the specification relation should be as wide as possible for a given input x. But incompleteness is the extreme of under-specification: if *any* result is acceptable on input x then the specification is incomplete at x, which is also undesirable.

2 Formal Methods

We examine several kinds of formal, mathematical specification and discuss the way in which state enters their notions of correctness. It is revealing to consider a simple archetype program whose state records data from previous inputs for subsequent use. Such a program models the use of a permanent database. It has 'store' inputs that modify the state with no significant output value, and 'find' inputs that retrieve previously stored values and output them. For example, in an address-book-like application, the state might be name-address pairs. Should the input be `find John Smith`, the output should be `sm137255@aol.com` if the state contains that pair, or output `not found` if the state doesn't have any pair with such a first element. Should the input be `store John Smith: smith27@ucg.ie`, the output should be `OK` if no state pair has a first element for that person, or `duplicate--ignored` if one does.

This sloppy specification could be developed into a precise relation S_a using explicit state as described above.

Or, a state-hidden specification $S_a{}'$ might be something like the following:

> Suppose $t \in D^n$ is an input sequence $t = (x_1, x_2, ..., x_n)$ that begins in the reset condition. There are two cases exemplified by:
> (i) x_n is `store John Smith: smith27@ucg.ie`. Then the output is `OK` if the name does not appear with `store` in any earlier member of the sequence t, or `duplicate--ignored` if it does;
> (ii) x_n is `find John Smith`. Then the output is `sm137255@aol.com` if the earliest member $x_i \in t$ involving the name "John Smith" and "store" is (say) `store John Smith: sm137255@aol.com`, or `not found` if there is no such x_i.

In the subsections to follow, descriptions of each formalism are brief and imprecise, intended only to remind the reader of their properties.

2.1 Mills's Functional Calculus

Harlan Mills proposed a 'program calculus' in the 1980s, which has elements drawn from a Turing-like operational semantics and also elements of denotational semantics. In its most complete form [9], his ideas were applied to a subset of Pascal. A denotation meaning M is assigned to a program P beginning with its elementary statements and proceeding inductively to composite statements. The end result is

to assign to P a function $\boxed{P}$ that can be represented as a concurrent assignment statement,

$$(X_1, X_2, ..., X_n) := M(X_1, X_2, ..., X_n),$$

where the X_i are the variables that occur in P, and the concurrent assignment reflects how the program changes their values. Mills takes specifications to be functions S mapping the values of program variables to the same, and defines correctness as $S \subseteq \boxed{P}$. Thus the program may manipulate variables not in S as it chooses and extend S domains, but is constrained on variables and domains covered by S.

For our purposes, a program P in Mills's formalism may be thought of as having just three variables: X_i for input, X_r for output, and X_h for state. The functions f and g for P are projections of $\boxed{P}$ onto the value sets of X_r and X_h respectively.

Interesting specifications are maps from the input- and state-value sets to the output-value and output-state-value sets. For example, since the archetype specification S_a of Section 2 is a function, it can serve as a Mills specification. If a program P is written in the obvious way to implement S_a, setting X_h for 'store' inputs, and examining it for 'find' inputs, it will be possible to prove P correct wrt S_a.

State enters a Mills proof of correctness in the form of program-supplied constraints on the state variables. In trying to demonstrate the functional containment of the definition, it will be very helpful to reduce the number of cases to be considered to just those state values that the program actually allows to occur. Insofar as this is done perfectly, the correctness proof will be effectively handling only the cases corresponding to sequence correctness; on the other hand, if the actual states are ignored, the proof will look more like state-blind correctness. In practice, any proof will be a compromise between how difficult it is to prove extra cases involving state, and how difficult it is to express the state constraints so that they simplify the proof.

2.2 Hoare Logic

Following ideas of Floyd [3], Hoare devised a first-order program logic [5] to specify what a program should or does do. The predicates of this logic range over the value sets of variables used in a program. Suppose these to be $(x_1, x_2, ..., x_n)$, and abbreviate this n-tuple as $\mathbf{X}$. The statement:

$$P(\mathbf{X})\{C\}Q(\mathbf{X}) \tag{1}$$

asserts that should statement P hold of the variable values before the execution of program C, then statement Q will hold of them afterward. P is the *precondition* and Q the *postcondition*. In Q, variables are 'primed' to refer to their values before execution of C, unprimed variables referring to after execution. With some convention about what variable(s) are the input and output, equation (1) constitutes a specification of an unknown program C. On the other hand, if a particular C is given, the assertion (1) either holds or does not, and C is correct wrt the

specification if it does. Hoare provides a proof method for imperative programs in which preconditions and postconditions for individual statements are determined, and rules of inference allow these to be combined until the desired P and Q are reached for the whole program (or not reached, if the program is not correct).

Again, let there be just six variables in the assertions, x_i for input, x_r for output, x_h for state, and primed x_i', etc. in postconditions.

In general, the Hoare proof method need not precisely constrain the functions f and g for an arbitrary program C. The Hoare rules of inference for certain program statements like loops and recursive program calls include arbitrary invariant predicates that need not capture completely the effect of these statements. The rules therefore determine a predicate $R(\mathbf{X})$ such that $true\{C\}R(\mathbf{X})$. When viewed as a correctness relation, R constrains the semantic maps f and g of C, but might not determine them. That is, for example, $f(x_i, x_h) = x_r \Rightarrow R(x_i', x_h', x_r)$, but the reverse implication does not necessarily hold. Similarly, the pre- and postconditions constitute a relational specification $S(\mathbf{X}) = P(x_i, x_h) \Rightarrow Q(x_i', x_i, x_h', x_h, x_r)$.

State enters a Floyd/Hoare proof through the notion of invariants. An invariant is a statement $I(\mathbf{X})$ introduced to ease the proof of correctness by strengthening the precondition. If the proof of $P(\mathbf{X})\{C\}Q(\mathbf{X})$ cannot be accomplished (perhaps only because the human or mechanical prover is not up to the task), it may be that $P(\mathbf{X}) \wedge I(\mathbf{X})\{C\}Q(\mathbf{X}) \wedge I(\mathbf{X})$ is easier to prove, and is equivalent if in addition $I(\mathbf{X})$ can be shown to hold initially. Invariants often describe how particular variable values, notably those of state, behave. Introducing invariants is the creative part of a proof of correctness, since nothing constrains their form except the need to simplify the proof. A strong invariant introduces a state restriction that makes the rest of the proof easier, but then establishes a difficult proof obligation of its own.

For the specification S_a, the postcondition is a straightforward assertion containing cases such as: if x_i' is a 'find' and x_h' contains the name, x_r contains the address and $x_h = x_h'$; or, if x_i' is a 'store' and x_h' does not contain the name, then the name and address are added to x_h and x_r is the 'OK' message. An invariant that might help with the proof is that x_h has no duplicate names.

2.3 Z Specification

The Z specification language, developed by the Programming Research Group at Oxford University based on seminal work by Jean-Raymond Abrial, is an example of a model-oriented specification language, which became an international standard in 2002 [1]. It is a formal notation based on set theory and first order predicate calculus [4]. A specification in Z describes an abstract state space, often using a predicate over the state space (an invariant) to describe valid states. An operation over the state space is given using a predicate P_{op} over input variables $X?$, output variables $X!$ and variables of the 'before' state X_h and 'after' state X_h'. The predicate describes conditions on the inputs and before states, the possible outputs, and how

[4] A 'formal method' strictly refers to a scheme for software development in which a formal, mathematical specification plays a central role. It is easy to mistakenly call the mathematical notation a 'formal method.'

the state changes from before to after.

For example, the state space for the specification S_a could be modeled as a partial function from names to addresses.

AddressBook

$$aBook : PERSON \pfun ADDRESS$$

Note that, in this example, the invariant (True) is omitted, meaning that any partial function from *PERSON* to *ADDRESS* is a valid address book.

The Store operation should check if the name is in the domain of the state function and add the new maplet if it is not and output OK, or return an error if it is.

StoreOK

$$\Delta AddressBook$$
$$p? : PERSON$$
$$a? : ADDRESS$$
$$r! : REPORT$$

$$p? \notin \mathrm{dom}\, aBook$$
$$aBook' = aBook \oplus \{p? \mapsto a?\}$$
$$r! = OK$$

StoreError

$$\Xi AddressBook$$
$$p? : PERSON$$
$$a? : ADDRESS$$
$$r! : REPORT$$

$$p? \in \mathrm{dom}\, aBook$$
$$r! = DuplicateIgnored$$

The Find operation should check if the name is in the domain of the state function and output the result if it is, or return an error, in both cases leaving the state unchanged. The formal schemas are omitted.

Although a Z specification describes exactly what states and outputs occur for given inputs, it says nothing about the sequence of operations (and hence states) which might occur. However, a specification will normally specify an initial state, together with a proof that it is a valid state, i.e. satisfies the state invariant. An initial state for the address book S_a would be the empty function, i.e. containing no maplets. Since $\emptyset$ is a partial function of the required type, by default it satisfies the invariant '*True*'.

InitAddressBook

$AddressBook'$

$aBook' = \emptyset$

The system is considered to be modeled by the initial state followed by an arbitrary sequence of operations. Further, there is a proof requirement on each operation that the change of state will always result in a valid state, given a valid state as input.

An implementation of a Z specification will include a concrete representation of the state space, and for each operation in the specification, a corresponding concrete operation over the concrete states. Correctness is then a matter of demonstrating that each abstract operation is refined by its corresponding concrete operation, $Op_A \sqsubseteq Op_C$. If we consider Op_A and Op_C as relations over $(D \times H \times H \times R)$, this means proving that: $\forall\, X? \in D, X_h \in H$,

$$(\exists\, X_h' \in H, X! \in R, (X?, X_h, X_h', X!) \in Op_A) \\ \Rightarrow (\exists\, X_h' \in H, X! \in R, (X?, X_h, X_h', X!) \in Op_C)$$

and

$$\forall\, X_h' \in H, X! \in R, (X?, X_h, X_h', X!) \in Op_C \Rightarrow (X?, X_h, X_h', X!) \in Op_A.$$

This Z notion of correctness falls somewhere between state-blind correctness and sequence correctness. It is not so strong as state-blind correctness, because it can be assumed that the input state is valid, and hence so also is the output state. It is not the same as sequence correctness because a proof is required for all *valid* input states, even those that might never result from any sequence of inputs.

3 Composition of Components

In considering the construction of software systems from components described by our general formalism, the descriptions of several functions and specifications will be involved. We distinguish these by subscripts on mappings, relations, etc. The mechanism of system construction will be restricted to 'series' combination, that is, one program's output becomes the input to the next [5]. This mechanism does not capture all cases of system construction, but it it evidently an important one and within it the important theoretical issues arise.

3.1 Defining Composition Formally

The proper intuitive meaning of persistent state for systems in which components with state are combined is not evident. On the one hand, system state could be taken to be 'global' in that all components might access and modify it; or, it could

[5] Series invocation is *not* the subroutine mechanism of one program 'calling' another, which has certain complicating aspects better left out of a beginning theory [10].

be 'private' in that within the system, components can use only their own state sets. The model of object-oriented classes designed using information hiding suggests the latter choice.

Let two components A and B be placed in series to form system U as shown in Fig. 1. We want to define the actions of the system code in terms of those of the

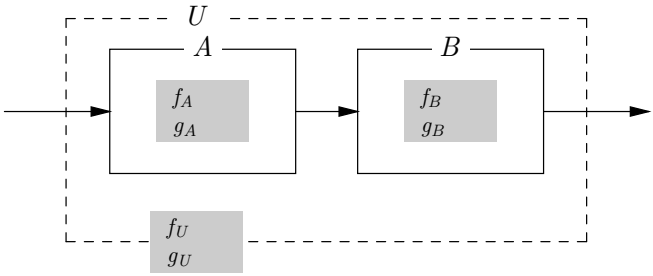

Fig. 1. Series combination of two components into a system

components.

Definition 3.1 U's input domain set is $D_U = D_A$, its output domain set is $R_U = R_B$, its state set is $H_U = H_A \times H_B$, and its two mappings are:

$$f_U(x, (h_1, h_2)) = f_B(f_A(x, h_1), h_2), \tag{2}$$

and

$$g_U(x, (h_1, h_2)) = (g_A(x, h_1), g_B(f_A(x, h_1), h_2)), \tag{3}$$

for all $x \in D_U, h_1 \in H_A, h_2 \in H_B$.

In Definition 3.1 the functional mapping f_U of equation (2) is a straightforward composition. The state mapping g_U in equation (3) carries an input and a pair of state values for A and B into a pair, the first element being what A does to its state, the second element what B does to its state, but in the latter, B sees an input that is the output of A.

Definition 3.1 reflects the 'compositionality' of the functional component formalism, because it captures the system behavior as that of a larger 'component.'

It is the natural intent of placing A and B in series that $R_A \subseteq D_B$; however, if this is not the case and for some $v \in D_A$, $f_A(v, h) \notin D_B$, then f_U is undefined at (v, h) and for U to be correct v must be a don't-care input of 'component' U.

The reason for our choice of state private to each component rather than global to a system is apparent in equations (2) and (3). With private state, the state functions g do not interact, not even by composition, which considerably simplifies the theory. Keeping the state within each component also allows the component developer to analyze g without reference to any other components.

It is more difficult to find an intuitively correct definition for the way in which component specifications ought to combine into a system specification. One possibility is to take the composition of relations:

Definition 3.2 (*Straightforward composition*)

$$S_U = S_A \circ S_B$$
$$= \{(x, (h, h'), (k, k'), r) \mid \exists\, y, (x, h, k, y) \in S_A \wedge (y, h', k', r) \in S_B\}.$$

Unfortunately, straightforward composition does not properly handle don't-care cases, as the following example shows [6].

Example 3.3 Let components A and B be placed in series to form system U. Let their specifications include the quadruples:

$$(x_0, \_, \_, e_1) \in S_A \text{ and } (x_0, \_, \_, e_2) \in S_A.$$

$$(e_1, \_, \_, z) \in S_B \text{ but } e_2 \text{ is a don't-care input in } S_B.$$

For the code,

$$f_A(x_0, \_) = e_2 \text{ and } f_B(e_2, \_) = z' \neq z.$$

A concrete version of this situation occurs when A produces an error message on input x_0 whose particular form might be either e_1 or e_2. However, e_2 is so unusual that it is a don't-care input for B.

The specification quadruples of Example 3.3 are not inconsistent with A and B being correct wrt S_A and S_B. However, using straightforward composition to define S_U, $(x_0, (\_, \_), (\_, \_), z) \in S_U$, but $(x_0, (\_, \_), (\_, \_), z') \notin S_U$, while $f_U(x_0, (\_, \_)) = z'$, so the series combination is not correct.

The definition of composition can be adjusted to handle the example by removing from the composition any quadruples for which there is an undefined alternative:

Definition 3.4 (*Strict composition*)

$$S_U = (S_A \circ S_B)$$
$$\setminus \{(x, (h, h'), (k, k'), r) \mid (x, h, h', y) \in S_A \wedge \forall\, t, (y, h', k', t) \notin S_B\}$$

In Example 3.3, x_0 is a don't-care input in the strict-composition specification S_U, which makes S_U vacuously correct. However, it is intuitively unlikely that x_0 was intended to be a don't-care input for U, so strict composition seems to hide a difficulty that might be better exposed. The creation of vacuous cases indicates an information loss in the system specification. The best intuitive description of Example 3.3 seems to be that A should not have been placed in series with B, because the system behavior on input x_0 is unexpected.

Only the strict-composition definition is acceptable—we cannot have composition of two correct components producing an incorrect system. Yet straightforward composition is a better basis for intuition. A resolution of the definitional dilemma

[6] Since state does not play a role in the example, the state values are shown only as place holders.

could be that a proposed component combination be evaluated to see if the definitions disagree. If they do, the combination could be considered to be dangerous. However, this natural idea has disturbing ramifications, because it calls into question the 'composability' of formal specifications. The danger in a combination is entirely a 'system' property that depends on both components; nothing in their individual specifications can predict it.

3.2 Composition in Existing Formalisms

The Mills, Floyd/Hoare, and Z formalisms were devised to describe complete systems. Each has a notion of 'composition' within a system, but none matches well with the component idea of series combination. Part of the difficulty is the formalisms' attempt to incorporate every programming notion within their own framework rather than to externalize the idea of combination. Thus Mills's calculus and Hoare logic, for example, include two kinds of composition, by juxtaposition (statement sequence within a program) and by procedure call, but neither is like series combination of components, particularly in the way state is treated. Components in series intuitively 'invoke'[10] rather than call each other—control leaves the first for the second, never to return.

3.2.1 Mills's Calculus

Consider two programs and their specifications, captured in Mills's formalism as $\boxed{P_A}$ (specification function $S_A \subseteq \boxed{P_A}$ for correctness) and $\boxed{P_B}$ (specification S_B), and their combination into a system U. By properly renaming the input, output, and state variables of P_A and P_B, $\boxed{P_U}$ can be described in terms of the component descriptions. (Identify the output variable of P_A with the input variable of P_B and keep their state variables distinct.) Since the specifications are functional, straightforward composition, $S_U = S_A \circ S_B$, does not display the difficulty with non-determinism in Example 3.3.

3.2.2 Floyd/Hoare Logic

As in the Mills calculus, by renaming the input, output, and state variables of the two components A and B of a series system U, the precondition for A and the postcondition for B can be brought to common terms as pre- and postconditions for U, defining a natural S_U. Unfortunately, the properties of S_U are less than desirable. First, although it is natural to expect that the precondition for B should be a logical consequence of the postcondition for A, this condition is not necessary to the correctness of U. For example, it could happen that the system postcondition is so weak that B's precondition is not needed. Second, the non-determinism displayed in Example 3.3 can easily allow the components to be correct yet the system not correct. Treating the component pre- and postconditions as relations, their strict composition according to Definition 3.4 can eliminate this difficulty, but there seems to be no natural expression of strict composition in the pre- and postcondition form.

Object-oriented programming uses Hoare specifications (which it calls 'contracts') for its operations (which it calls 'methods') [7]. The reuse of object-oriented classes as components [8] is based on conventional programming-language procedure calls, not invocations, so the match with our theory is not close.

3.2.3 Schema Composition in Z

The Z specification notation includes a notion of schema composition, representing the effect of one operation followed by another. However, there are some important differences between the Z schema composition, and specification composition in this paper. Given two Z operations, Op_A and Op_B, the composition $Op_A \fatsemi Op_B$ is defined only when the operations are both defined over the same state space—the state is global. The composition is sequential, in that the output state of Op_A becomes the input state of Op_B, but the inputs and outputs are shared.

More formally, if Op_A is a relation in $D_A \times H \times H \times R_A$, and Op_B is a relation in $D_B \times H \times H \times R_B$, then $Op_A \fatsemi Op_B$ is a relation in $(D_A \cup D_B) \times H \times H \times (R_A \cup R_B)$.

The choices made in defining schema composition in Z come from an earlier time in which the modern view of components was not anticipated.

4 Imperfect Specifications

Software development using formal mathematical methods is usually imagined to proceed as follows:

(i) Devise a specification. That is, produce the relation S that a program to be written must satisfy, or whatever variation on S a particular formalism provides.

(ii) Write a program P to satisfy S. Sometimes the formalism provides a way to manipulate S to obtain code that is necessarily correct; at the other extreme, code may be written to satisfy an intuitive understanding only loosely connected to S.

(iii) Prove that P is correct wrt S. If P is not correct by construction, this step may be very difficult.

Experience with this paradigm shows that S is almost always deficient [4]. Formal specification is a demanding, complex activity for which most human beings are ill-suited. If P is constructed from S, the deficiencies may be hidden until P is tested or used; if P is separately constructed by hand, the proof fails, sometimes because P is not as desired, but more often because S is wrong. Despite this almost universal experience, there has been little explicit study of specification errors. In component-based software development, component specifications and code are a given, not usually subject to adjustment. This setting is thus ideal for a study of specification mistakes and the way they enter into composition. By considering what can go wrong, we hope to understand the special specification needs of CBSD.

4.1 Example 3.3 Revisited

Restating the problematic example from Section 3.1: Components A and B are placed in series, A produces an error message on input x_0 whose particular form might be either e_1 or e_2; e_2 is a don't-care input for B.

There are a number of possibilities where either specification S_A or S_B is deficient:

- It might be that S_A is wrong at x_0—perhaps there should not be an error message or e_2 is not an acceptable error message for x_0. The solution might be to remove from S_A the tuple relating x_0 to e_2, thereby necessitating changing the component code A accordingly.
- In the case S_B is really incomplete and e_2 should not be a don't-care input, then a good solution might be to treat e_2 like e_1. Both specification S_B and component code B will need to be changed in this scenario.
- Alternatively, perhaps S_B is wrong at e_1 and requiring a particular result is overly prescriptive. In this case, we could make e_1 a don't-care input in specification S_B, in which case the component B will not need to be changed.
- Another case is that S_A is wrong at x_0, and that e_1 is not an acceptable error message. Now, component A is OK, even though its specification is wrong.

These possibilities show how fragile the combination of imperfect specifications for components can be, and the difficulty of sorting out the cause and required changes when something untoward appears at the system level.

4.2 Substitution and Modification of Components

If component-based development is to realize its promise, it must be possible to synthesize specifications of component combinations from specifications of the components. This makes strong new demands on the component-specification formalism. Developers expect that they can modify components—both code and specification—in isolation, reason about the modified components in isolation, and then substitute the changes into systems. The following are minimal expectations of this process:

(i) If component code is changed yet is still correct wrt an unchanged component specification (so-called "perfective maintenance"), a substitution should not affect any system correctness properties.

(ii) If a component specification is extended in the sense that all of the original tuples still belong to it but others are added, using the extended specification should not do more than extend a system specification.

(iii) Mistakes in components should not be hidden at the system level.

Example 3.3 will violate expectation (ii) if it is viewed as extending a first component specification to allow message e_2.

It is easy to construct other pathological examples of situations familiar to practical developers, such as:

- Each of two specifications is wrong, code for them both is correct, yet their composition is not wrong, violating expectation (iii) above. (This can occur if wrong output values from the first component are corrected by the second; and also if wrong values from the second component never arise because the first has a restricted range that does not excite them.) The danger in this situation is that the intuitive property of the system is not derived from the components, but arises accidentally.
- A composition is correct, despite its first component being wrong (perhaps because the wrong output is corrected). When the first component is corrected the composition fails (perhaps because the second component cannot handle the correct first-component output).

From such examples we learn that apparently harmless specification mistakes at the component level introduce uncertainties that make system-level reasoning suspect. The perfection required of the formal-specification development model is the culprit.

5 Conclusions

We have re-examined functional and relational formal specifications taking explicit account of persistent state, and applied these to software components. The exercise shows that formal specification was largely conceived and developed as a single-program idea. It imagines that program and specification are brought into the relationship of correctness and that is the end of it. On the contrary, in the component view of software development correctness at the component level is only the beginning of a process of reasoning about combinations of components into systems. It is often said that mathematical specification methods have 'composability,' where (say) testing methods do not. This insight rests on the obvious notion of functional composition, but it is called into question when we consider human mistakes in the pieces being composed. Such mistakes combine in strange and wonderful (or terrible) ways and it is little help that these ways can be worked out after the fact. If there is to be a formal theory of component combination into systems, desirable system properties must be obtained from properties of the components, and we have shown this to be problematic. The very attributes that go into making a good formal theory for stand-alone systems may be counterproductive for component-based systems.

Non-determinism is a good example. Specifications that are no more prescriptive than necessary are viewed as a good thing for a single program. By allowing non-deterministic choices (that is, the specification is a relation that is not a function) the implementor is given useful freedom, and correctness proof is simplified. But non-determinism is a bad thing in a component specification, because it forces an unnatural definition of composition, and it behaves badly when people make specification mistakes. The Mills formalism is entirely functional, while the Floyd/Hoare formalism is preferred because it allows non-determinism. For component theory the choice might be reversed.

Parnas and Madey recognise, in [11], that the requirements document for a component will not fully describe the behaviour of that component. There may be many observably different components that satisfy the same requirements. They suggest that a *Software Behaviour Specification* may be required, which describes the actual (functional) behaviour of the component. This agrees with our observation that non-determinism may be good for system specifications, but perhaps not for describing the behaviour of a component.

If state specifications are non-deterministic, or in the hidden-state case if state is not specified at all, combination of wrongly-specified components can be particularly unpredictable. The unexpected effects occur only when particular states arise, and it is very difficult to establish if this ever in fact happens. Proofs at the component level may be state-blind, so that they never come to grips with which states actually occur.

The unavoidable specification mistake, given the vagaries of human ability, is incorrectness—a person intends something different than the formal mathematics defines. Critics of formal methods say that formalism merely shifts the problem of incorrect implementation to wrong specification. The counter argument is that the shift is worthwhile, since mistaken specifications are easier to detect and understand than mistaken code. In the component setting the critics' side of this debate is strengthened. Since a component developer cannot know the application to which his component will be put, nor the other components with which it will be used, he cannot identify likely specification mistakes—they will become important only at the system level for some unimagined system.

References

[1] 13568:2002, I., *Information technology—Z formal specification notation—syntax, type system and semantics*, international Standard.

[2] Abadi, M. and L. Lamport, *Conjoining specifications*, ACM TOPLAS (1995), pp. 507–534.

[3] Floyd, R. W., *Assigning meanings to programs*, in: *Proceedings Symposium Applied Mathematics* (1967), pp. 19–32.

[4] Gerhart, S., D. Creigen and T. Ralston, *Observations on industrial practice using formal methods*, in: *15th ICSE*, 1993, pp. 24–33.

[5] Hoare, C. A. R., *An axiomatic basis for computer programming*, Comm. of the ACM (1969), pp. 576–585.

[6] Li, H., S. Krishnamurthi and K. Fisler, *Verifying cross-cutting features as open systems*, in: *Proc. SIGSOFT FSE*, 2002, pp. 89–98.

[7] Meyer, B., "Object-oriented Software Construction," Prentice Hall, 2000.

[8] Meyer, B., *The grand challenge of trusted components*, in: *Proc. ICSE 25*, 2003, pp. 660–667.

[9] Mills, H., V. Basili, J. Gannon and D. Hamlet, "Principles of Computer Programming: A Mathematical Approach," Allyn and Bacon, 1987.

[10] Parnas, D., *On a "Buzzword": Hierarchical structure*, in: *Proc. IFIP Congress* (1974), pp. 336–339.

[11] Parnas, D. and J. Madey, *Functional documentation for computer systems engineering*, Science of Computer Programming (1995), pp. 41–61.

[12] Szyperski, C., "Component Software," Addison-Wesley, 2002, 2nd edition.